REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
October 7, 1995

3. REPORT TYPE AND DATES COVERED
Quarterly Technical Report

4. TITLE AND SUBTITLE
Quarterly Technical Report
Massive Data Analysis Systems

5. FUNDING NUMBERS
F19628-95-C-0194

6. AUTHOR(S)
Richard Frost Mike Wan Vibby Gottemukkala
Chaitanya Baru Reagan Moore Anant Jhingran
Richard Marciano Joe Lopez

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
San Diego Supercomputer Center
P0 Box 85608
San Diego CA 92186-85608

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ARPA/ITO
Att: Charlene Veney
3701 N. Fairfax Drive
Arlington VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N/A

11. SUPPLEMENTARY NOTES

19961023 295

12a. DISTRIBUTION/AVAILABILITY STATEMENT
ARPA/ITO
ESC/ENS
DTIC
ARPA Technical Library

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The creation of a Massive Data Analysis System (MDAS) will enable new modes of science through improved data management of scientific data sets. This requires a scalable software infrastructure that can manage petabytes of data, support rapid access of selected data sets, and provide support for subsequent computationally intensive analyses. To accomplish this, object-relational database technology is being integrated with archival storage systems. By supporting transportable methods for manipulating the data, it then becomes possible to analyze selected data sets on remote systems. The MDAS becomes a scheduling system, managing the flow of data and computation across distributed resources. Usage models are needed that simplify the identification, transport and analysis of large collections of data. The system must automate the collection of metadata describing the data set attributes, and handle interactive WEB access, distributed database access, and discipline specific application interfaces. A software infrastructure has been designed which manages user access restrictions, matches application requirements with resource availability, and schedules the data movement and application execution. Development of this software system is proceeding on schedule, with selected applications testing the initial prototypes.

14. SUBJECT TERMS
Scientific Data Mining

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	

QUARTERLY TECHNICAL REPORT

June 1996 – August 1996

Massive Data Analysis Systems

San Diego Supercomputer Center

Reagan Moore, Principal Investigator
Chaitanya Baru, Richard Frost, Richard Marciano, Mike Wan

IBM T.J. Watson Research Center

Vibby Gottemukkala, Anant Jhingran

Sponsored by: Advanced Research Projects Agency CSTO

ARPA Order No. D007 and D309
Issued by ESC/ENS under Contract F19628–95–C–0194

Distribution:

ARPA Agent ESC/ENS Bldg 1704, Rm 114 5 Eglin Street Hanscom AFB, MA 01731-2116	(2 copies)
ARPA/ITO Attn: Charlene Veney 3701 N. Fairfax Drive Arlington, VA 22203-1714	(2 copies)
ESC/ENK Attn: Ms Carole Stephan Bldg 1704, Rm 119 5 Eglin Stree Hanscom AFB, MA 01731-2116 (Letter of Transmittal Only)	(letter)
Defense Technology Information Center Cameron Station Alexandria, VA 22034-6145	(2 copies)
ARPA/Technical Library 3701 N. Fairfax Drive Arlington, VA 22203-1714	(1 copy)

Source:

Richard Frost
San Diego Supercomputer Center
P.O. Box 85608
San Diego, CA 92186-9784

Contents

ABSTRACT

The creation of a Massive Data Analysis System (MDAS) will enable new modes of science through improved data management of scientific data sets. This requires a scalable software infrastructure that can manage petabytes of data, support rapid access of selected data sets, and provide support for subsequent computationally intensive analyses. To accomplish this, object-relational database technology is being integrated with archival storage systems. By supporting transportable methods for manipulating the data, it then becomes possible to analyze selected data sets on remote systems. The MDAS becomes a scheduling system, managing the flow of data and computation across distributed resources. Usage models are needed that simplify the identification, transport and analysis of large collections of data. The system must automate the collection of metadata describing the data set attributes, and handle interactive WEB access, distributed database access, and discipline specific application interfaces. A software infrastructure has been designed which manages user access restrictions, matches application requirements with resource availability, and schedules the data movement and application execution. Development of this software system is proceeding on schedule, with selected applications testing the initial prototypes.

Chapter 1

Task Objectives

The overall objective of the Massive Data Analysis System is the construction of a scalable system that integrates data management with computation to support analysis and assimilation of arbitrarily large data sets. The main goals for the last four quarters have been the development of consensus on (a) application requirements, (b) hardware and software systems that can meet the application requirements, (c) and the application presentation interfaces needed to make the system accessible to researchers.

These goals are best evaluated through the implementation of prototypes. Hence a continuing objective is the development of a sequence of prototypes to test the efficacy of the software design for WEB access, application access, and data handling system interoperation. In order to provide a mechanism for technology transfer, a final objective is to involve a commercial software vendor in the development of a scalable system that will be able to handle arbitrarily large data set sizes.

Integrated data and object handling systems have the potential to be the next major software infrastructure in the evolution of computer systems. If designed correctly, the data handling system can support a uniform user interface to distributed heterogeneous systems needed for global computing. The data handling system acts as a scheduling system that resides above the operating system. Users interact directly with data attributes, relying on the data handling system to locate the data, move the data to appropriate compute resources, and execute applications on the data sets.

Chapter 2

Technical Problems

The primary challenges in the MDAS project are (a) the integration of data management systems with archival storage and (b) end-user solutions for the replacement of the (Unix) file paradigm with a higher-order interface to data, methods, and resources.

At project onset, database management systems (DBMSs) and hierarchical (archive) storage systems (HSSs) were not interoperable. Replacements for the Unix OS file paradigm existed, but only on single-user systems and largely without high-order interfaces to resources. In order to develop concrete extensible solutions with reasonable technology transfer attributes, we have taken a two-prong approach to these problems. First, we rely heavily on application prototypes to investigate the viability of possible solutions to the primary challenges. Second, we continue to carry out an in-depth research into the design and specification of production-grade software solutions. A number of technical problems have emerged from these efforts which we will address throughout the remainder of the project:

1. Hierarchical storage system (HSSs) have a limited number of I/O channels and physical read/write heads for archival (tape) storage. Therefore, an application with large resource requirements can easily monopolize the entire storage system unless a reasonable resource management policy is implemented in the HSS. This amounts to implementing a queuing system within the HSS to mitigate requests from multiple clients.

2. HSS software interfaces contain general I/O routines such as read, seek, rewind, etc. However, most HSS application client interfaces limit data transactions to file get and put—primarily to keep applications from monopolizing resources. For example, an application controlling a tape drive with seeks and rewinds for 2 hours leads to poor resource utilization for the general user population. This means that when an HSS client wants to read a large file, it must have either (a) enough local memory (RAM or Disk) to read the file in total, or (b) access to an intelligent spooler which can buffer the read and make incremental calls to the HSS for large file blocks. The latter case is most common.

3. The development of an MDAS software infrastructure for general I/O interoperability between local and remote resources requires (a) library interfaces to individual

resources, (b) a high-level interface to hide individual resource semantics from the users, and (c) a high-level interface that supports multiple I/O paradigms across each supported resource. Not all platforms will have suitable library interfaces for all resources; e.g., a desired DBMS, HSS, or HTTP library might be missing. Hence, daemons must be developed to assist application clients on those platforms. Further, the build environment for the MDAS software can become arbitrarily complex without careful design considerations. Rather than "port" the MDAS software to each vendor hardware platform, the MDAS build environment should support compile-time configurable options for drivers to various resources.

Supporting a a high-level interface to hide individual resource semantics from users requires the development of intermediate buffering mechanisms just below the application level. For example, supporting a { read, seek, rewind } paradigm on top of a dataset opened (transparently) on an `ftp` resource will require local buffering and possibly remote re-reads of the file. A related issue is the support of legacy software systems; e.g., the use of Fortran unit numbers for I/O transactions on a DBMS large object, or the transfer of two valid file handles to a third-party data mover for a Unix-style "pipe" transaction.

Tension also exists between items "b" and "c" when a paradigm from one I/O source (e.g., DBMS query) does not match paradigms supported by another (e.g., Unix fifo pipe). Therefore, categories of I/O paradigms need to be identified and then supported by categories of high-level semantics.

4. Among the key resources that will be supported by MDAS are archival storage systems with a database system front-end. The database systems are used to store metadata and to provide access to the archive data sets. However, depending on the particular database system used, the implementation of this interface may be different. In order to insulate MDAS methods from such implementation issues, the MDAS system must provide appropriate mappings from the standard file I/O interface used by the methods to the actual interface supported by each database system front end.

5. Application clients, DBMSs, and HSSs all have response time limitations for I/O and general communication transactions. Coupling these systems requires careful design considerations to avoid request timeouts and blocked (hung) communication requests. For any particular component in the system, it is worthwhile to know in advance that another component is off-line—rather than to wait on an internal (possibly long) connection timeout condition.

6. Application clients, servers, DBMSs, and HSSs all have various authentication mechanisms which can vary among sites. In a distributed environment, it is often desirable to transfer methods and/or datasets from one resource to another for more efficient processing. This capability requires authentication interfaces between coupled systems (e.g., a tightly coupled DBMS and HSS) plus third party authentication mechanisms to permit a server to transfer a client's request to an external (third party) processing system.

7. Object-oriented (OO) software technologies greatly simplify the task of software engineering and hold great promise for software reuse. However, present-day OO compilers do not produce high-performance executables which is of paramount importance to

this project. Hence, MDAS implementations should choose application-efficient languages while providing interfaces to OO language semantics.

8. Applets (as implemented in the Java language) and other interpreted methods extend the OO paradigm to remote resources. However, applets are extremely slow in processing scientific datasets or performing moderate iterative tasks in general. For applets to be truly efficient, "just-in-time" compilation methods are desirable on the target platform.

9. Traditional DBMS and FTP technologies rely on sequential I/O streams for transferring data objects. This approach has been demonstrated to give relatively poor performance unless aggressive caching strategies are developed. The concern is that focusing on just caching strategies will be inappropriate for the more advanced technology that is based on parallel I/O streams.

10. Scientific applications should be able to access data and cache it locally no matter where the data is originally located. This is equivalent to requiring a catalog or expert system with universal resource name (URN) capabilities.

11. Support for parallel I/O streams must be done within the context of emerging standards. This requires tracking the MPI2 IO effort which is examining issues related to message passing within a compute platform and I/O to external peripherals. Interoperability between MPI and non-MPI processes will require specialized software interfaces.

12. The design of appropriate experiments to test the capabilities of the proposed system requires independent testing of individual infrastructure components. This requires dedicating separate portions of the testbed system to HSS support and to database support. The result is that it will be possible to quantify the memory, disk cache, and I/O requirements independently for each system, and then quantify what the integrated system will need in terms of hardware resources to have adequate performance.

Chapter 3

General Methodology

Scientists have an ever-increasing need to store, access, and manipulate unprecedented quantities of data. Modern data manipulation requirements include searches for correlations in large data sets, incorporation of empirical data to improve the predictive capabilities of computational simulations, and mining of existing data sets to derive better input conditions for new calculations. High performance data assimilation environments [?] require new modes of operation to integrate data mining and supercomputing. These large-scale and national-scale problems strain our current infrastructure and motivate the development of massive data analysis systems.

In order to implement a system that can meet these requirements, mutually-scalable technologies are needed in parallel data-handling systems, computational servers, local area networks, and resource scheduling environments. Further, these system complexities need to be presented to users in a set of navigatable hierarchies. This latter criteria insures that a novice can have a high degree of success in using the system, while experts can achieve major advances in analysis and resource efficiencies.

Other efforts in this area have focused exclusively on scalable I/O [16, 4], high-performance computation [11], high-performance communication [8], digital libraries [14], or object-oriented software integration [10]. These are valuable efforts which will provide an experience base for future system integrations. However, in order to be successful, we need to address all aspects of of the "massive data analysis" problem. It is far more effective to design these new-generation systems from a first principles approach, while taking available technology into consideration.

3.1 Approach

The execution path of a computational request ρ can be viewed as a trajectory in a configuration space defined by users $\mathcal{U}$, data $\mathcal{D}$, methods (i.e., operations on data) $\mathcal{M}$, resources $\mathcal{R}$, and time $\mathcal{T}$. Call this space $\mathcal{C}$. It is a discrete space, with countably many elements in each dimension but a finite number of achievable states for any given request. A Turing machine is an example subset of $\mathcal{C}$.

In this framework, the execution of a program at any moment in time is characterized by *microstates* in $\mathcal{C}$ which detail binary values in registers, machine-level operations, hardware circuitry, etc. For the purposes of software engineering, it is practical to describe execution paths at a higher level(!). Hence, one can choose a coordinate basis in $\mathcal{C}$ which describe *metastates*:

$\mathbf{u} \in \mathcal{U}$	:	the user(s) who instigated the request
$D \in \mathcal{D}$	:	data content and data storage "formats"
$M \in \mathcal{M}$	:	algorithm and/or program characteristics
$R \in \mathcal{R}$	:	a description the hardware execution environment
$T \in \mathcal{T}$	:	relative system and wall clock times

For example: in the formulation of a request ρ to run a program on a Unix system, a user specifies the program name and possibly the name of some input data sets at the command line. The O/S (a method) then carries out the execution of ρ. The actual execution of ρ at the machine level might be very different than what the user intended in the initial formulation. Nonetheless, the request transits through a sequence of micro- and meta-states. When the program terminates, the O/S records some metastate information in the directory listing of any output files produced: e.g., $\mathbf{u}$ (the owner) and T (the time of storage).

Metastate information is commonly termed *metadata*. When a program produces output files, it might encapsulate some metadata information about the file format to ease the task of re-reading the files at a future date. It is typically the user's responsibility to maintain diaries of data lineage, appropriate methods for their application area, and the set of resources available for utilization.

Note that the performance of request ρ is a path integration over the states traversed by ρ in $\mathcal{C}$. Therefore: the optimization of ρ (or the design of a system to support the efficient processing of ρ) depends upon knowing states available to ρ in $\mathcal{C}$. Hence, both *lineage* and path *availability* are important pieces of information for the complete formulation of requests.

A scheduling system is an example of a system tool (method) that takes a specific request ρ and chooses a path in $\mathcal{C}$ to optimize some performance metric of either the system or the request. Unfortunately, current schedulers typically work under the direction of scripts (e.g., `at` and `NQS`). Therefore, the entire specification of ρ is not explicitly exposed to the scheduler and the ability of the systems to choose near-optimal solutions is limited.

In general, obtaining knowledge about available metastates in current systems is problematic at best. For example, most operating systems today only track coarse information about $\mathcal{U}$ and $\mathcal{T}$ when storing data sets. Data Base Management System (DBMS) products go a little farther, tracking $\mathcal{U}$, $\mathcal{T}$, $\mathcal{D}$, and some of $\mathcal{M}$. Methods often have explicit couplings with $\mathcal{D}$ and $\mathcal{R}$—but these are rarely mentioned outside user manuals. So modern computing currently deals with the automation of subsets of $\mathcal{C}$, rather than $\mathcal{C}$ as a whole.

3.1.1 Files and Programs vs. Data Sets and Methods

The "path discovery" problem is largely due to the interface paradigms we force on users—or that users have come to expect. The most recognizable of these is the *file* paradigm. We store data in files, programs in files, and packages or groups of files in *directories*. Some operating systems store files with a transparent *file header* which helps the O/S track methods available (applicable) to the data. Many applications store their data in an application-dependent way with file headers or other embedded tags describing some of available metastates in $\mathcal{C}$. A general extension of headers and tags are application independent *self-describing files* [5, 15].

Despite the pursuit of file-based systems by computer hardware and software engineers, users are generally interested in *data sets* and *methods*. For example, a common user task is to update a *document*, which may be in fact composed of one or more data sets. A common scientific computing task is to run a *simulation*, which is very likely a compound method operating on a compound data set—possibly stored on a parallel file system. Manual tracking of (a) which methods are applicable to which data sets and (b) the storage format for each data set does not scale well for users or data handling systems. Thus, it is a win for both the user and the efficiency of large scale systems to *promote the users internal data set paradigm* to the forefront of a "massive data analysis system" [2].

The data set paradigm is commonly supported by DBMS software. It is implemented by maintaining a "memory" of data types, structures, and lineage in *metadata*. However, DBMS metadata on methods is not maintained with the same rigor as data sets. As in Unix systems, the user is required to maintain a diary of operations available to given data sets. Another way to view this is that current DBMS language models do not support transitive operations on data. Suppose that document **d** is stored in format **a** but desired in format **c**. Further suppose that translations from $\mathbf{a} \rightarrow \mathbf{b}$ and $\mathbf{b} \rightarrow \mathbf{c}$ exist. Current DBMS's will be at a loss to automatically carry out $\mathbf{a} \rightarrow \mathbf{b} \rightarrow \mathbf{c}$ using standard data models.

The notion of "format" is very important to both parallel data storage and parallel methods on data. Continuing the above example, format **a** could very well be the description of a file striped on parallel disk, and format **c** could be the desired in-memory structure of a parallel application. For parallel (distributed memory) data-mining applications, discovery of transitive "gather-scatter" operations is imperative for performance. Otherwise, the application must take the conservative approach of serial agglomeration at the source and parallel redistribution on receipt. Without appropriate metadata or explicit instructions from a user, the application and file server are always forced into this mode.

Another important notion of format is the specification of numerical, textual, and higher-level media "types" within a document. Since data-mining applications are often 3rd-party readers of data, they need a description of the data set's *datatype(s)* to facilitate in-memory instantiation of the data. Without a datatype specification, the application is at a loss to do anything useful (other than copy) the data set. When datatype definitions are coupled with contextual information about a data set's content, a 3rd party application can perform meaningful operations on the data, including: extraction, reduction, and any user-defined analysis function.

All of these notions of data set format are subdimensions of $\mathcal{D}$. Extracting data subsets from a large storage object are simply projections in $\mathcal{D}$. A gather-scatter operation between parallel disk storage and parallel application memory is a transpose operation (from $\mathcal{M}$) in $\mathcal{R}$ and $\mathcal{D}$. Having metastate information available to all elements (with the correct access attributes) in a system increases the capability of doing useful work.

3.1.2 Resource Metastates

The concept of format extends well beyond data sets. A method can be considered to be formatted for uniprocessors, shared-nothing parallel processing, or general parallel computation (or all of the above). Note that these attributes are metastates of resources. Servers, schedulers, and even client applications who are "resource aware" can make efficient decisions about execution paths when the capabilities of available methods are exposed.

As a user's data access requirements grow, so does their interest in resources. Over time, tracking resources and maintaining knowledge about them becomes problematic for users. A notable success of Apple Macintosh computer [13] was the concept of *resource* metadata maintained by the operating system. This transparent storage of default metastates of documents and devices gave users a new paradigm and efficiency in computer interaction. (Unfortunately, the paradigm has not be significantly expanded in the last 10 years.)

As users' interests in resources grow, so do the accounting concerns of resource site administrators. A lineage of users' interactions with resources provides account administrators with information critical for the determination of resource service charges. User-resource lineage is also an aid to automated resource schedulers; for example batch request systems, run-queue software agents, and application-based schedulers [1]. Some expert schedulers find lineage useful for maintaining stochastic histories of resource utilization by users, methods, and data.

3.1.3 Metadata Messaging

One solution to these problems is to generalize system-level metadata support to include both lineage and path availability for all elements of a computational request. This would provide users of the system, software operating in the system, and system managers the critical information needed to optimize ρ. To support this need, a metadata definition is required that is

1. A natural extension of existing system-level metadata
2. General enough to encompass existing DBMS metadata conventions along with sequential and parallel "file" storage conventions.
3. Concise enough to be efficiently communicated in large numbers in distributed system environment

Items 1 and 2 are discussed in section 4.2, *Describing Metadata*. Item 3 is discussed in sections 4.2.2 and 4.3, *A Software Architecture For Metadata Messaging*. An overall presentation of MDAS system components is given in section 4.1. Section 4.4 discusses design specifications for application-level interfaces to the MDAS architecture. An example archival storage service is given in section 4.5.

Chapter 4

Technical Results

4.1 A Modular System Specification

A generic representation of a massive data analysis system is given in figure 4.1. The representation defines a system architecture showing the central role of a metadata repository in maintaining persistent metadata related to the state of the system. The metadata repository is essentially being used to prototype functionalities missing from current operating systems. It maintains metadata about users, data sets, methods, and resources known to the system.

The major entities shown in figure 4.1 are: *physical resources*, *MDAS server modules*, and *MDAS clients*. Physical resources are compute engines, visualization engines, storage systems, interactive systems which are "registered" in the MDAS system. Clients (DBMS client, application client, and Web client) make requests to the MDAS system. Server modules (DBMS server module, application server module, and Web server module) service client requests and are part of the Client Interface Services Layer described below. In addition, there may also be MDAS data sets which are also "registered" in the metadata repository.

In figure 4.1, the lines between the various components indicate request/control flow. For example, an MDAS application client may want to perform a computation on a given data set using a parallel compute engine and a visualization system. The requests from this client are sent to the appropriate MDAS application server module which is part of the MDAS Client Interface Services. The server module references the MDAS metadata repository to obtain information such as authorization, location, accessibility, availability of the various MDAS entities. The request is then passed to other MDAS services (e.g. Scheduling, Data Transfer) in order to schedule and process this request.

While the metadata repository in figure 4.1 is represented as a centralized resource, in the actual implementation each server module may choose to cache "hot" metadata records. For example, an MDAS entity typically has properties such as access restrictions, type, format, and location. An MDAS server module may cache this information for entities that it frequently references.

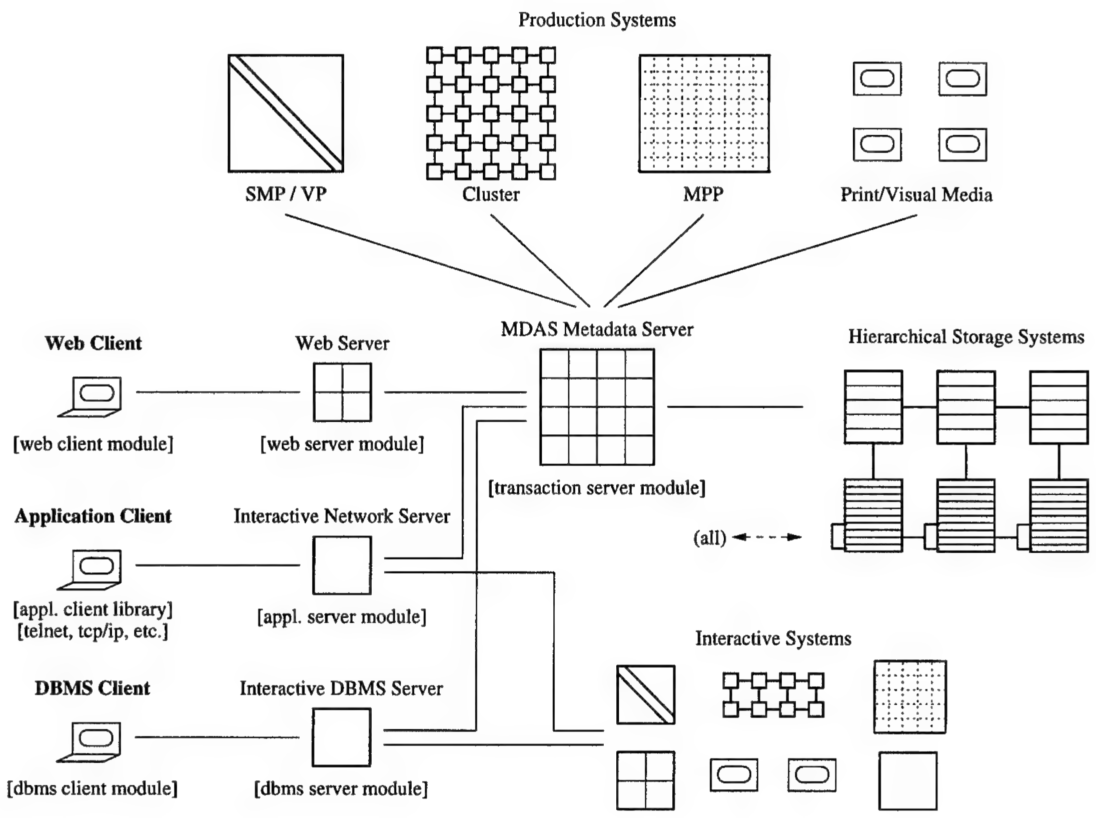

Figure 4.1: Massive data analysis system components. Software components (figure 4.4) are shown in [brackets].

One approach to constructing the MDAS *middleware* which allows clients and resources to participate in the MDAS system and service MDAS requests, is to implement site-specific daemons and software triggers to carry out the coordination of system requests. Another approach is to design the system middleware as a library of tools for the query, update, and storage of system level metadata. The software architecture that enables the building of such libraries is further described in section 4.3. From these tools, client and server modules can be built to provide efficient, coherent metadata transactions involving the metadata repository. In this framework, a set of site-specific resources are represented by a set of resource metadata records, thus allowing the MDAS system to schedule activities using this information.

4.1.1 MDAS Entities

Figure 4.1 shows three of the MDAS entities: clients, servers, and physical resources. Other MDAS entities include data sets and methods. Here we describe some of the characteristics of various MDAS entities.

Users/Clients

MDAS users may access the system from a variety of sources including Web access, direct access via an application program, and access via a DBMS. Users may be *anonymous* or *authenticated.* An authenticated user is one whose has previously "registered" with MDAS. MDAS provides mechanisms by which an anonymous user may use the Registration Services of MDAS to become a registered user. Non-registered users are treated as anonymous users and all anonymous users receive a default levels of access and service from MDAS.

Along with authentication information, users can be associated with level of service, level of access, and path histories. The level of service establishes the user's priorities and resource consumption constraints in using the system. Level of access controls the user's accessibility to resources in the system. Path histories are a cache of the user's pattern of usage of the MDAS system. They contain information on the frequency of access to data sets, resources, methods, etc. so that future user requests can be serviced efficiently.

Data Sets

Data sets are first-class MDAS entities. These include raw data sets stored on disk or on archival system (or both), data stored under various file systems, and data stored in DBMS's. In addition to standard information such as name, location, size, format, and access control, other information that may be associated with data sets includes partitioning (especially for parallel data sets) and structure (for typed data objects). As mentioned above, a data set is in the path histories of one or more users. This information is important in deciding when a data set needs to be "cached" (e.g. cached from archival store to disk), when it can be "swapped out", or when it needs to be replicated, etc.

Methods

Methods in MDAS include programs, macros, utilities, etc. which are used to operate upon and transform data sets in the system. As part of registering methods as entities in MDAS, the system will obtain and store metadata describing the method and its inputs and outputs.

Servers

In the category of MDAS servers we include computational and storage engines or resources. These entities have a given capacity and therefore the usage/consumption of these resources need to be monitored for effective scheduling. This includes compute engines (parallel and sequential), visualization engines, interactive systems, and disks. In addition, it is also possible to view a file server, DBMS, or archival storage system itself as an MDAS server.

4.1.2 MDAS Services

A variety of services are provided as part of the MDAS infrastructure. Figure 4.2 shows the various services. Each service requires access to the metadata repository. The repository along with the various services can be viewed as a single system, even though the services may be implemented by different modules/processes and the function of a particular service is implemented in a distributed fashion across several components/processes. The various services are briefly described below.

Registration Service

The registration service allows various MDAS entities to be registered in MDAS and results in the storage of metadata related to that entity in the repository.

Client Registration: By default, a client is treated as "anonymous" and, as such, is provided a default set of resources and default access within the system. Typically, a client would require greater access and capability than provided by default. The Client Registration Service is used to register a client within MDAS and to allow clients to request their desired level of service in the system. This allows MDAS to store metadata about level of service for each client and to use this metadata whenever it services a request from that particular client or class of client.

Server Registration: Server registration allows MDAS to be cognizant of the various servers and characteristics of servers that are available to the system. This, in turn, allows MDAS to process client requests by matching servers to requests for servers and scheduling servers in order to efficiently complete a set of requests. A variety of metadata can be associated with each server. In addition to a common set of metadata for all servers (e.g.

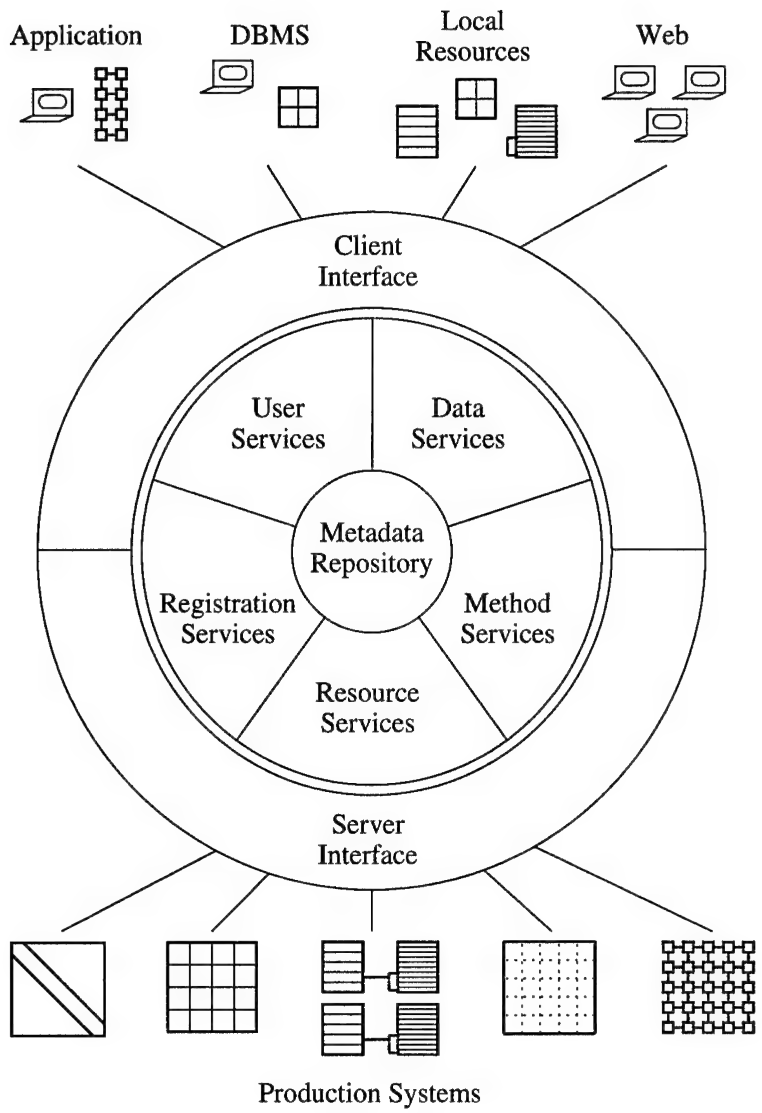

Figure 4.2: Services provided by MDAS infrastructure.

name, location), each class of servers can have their own metadata.

Data Set Registration: Access to data sets is one of the services that can be provided by MDAS. Similar to registering servers, one can register data sets in MDAS. This allows MDAS to directly control access to that data set by clients. This is different than registering a DBMS in MDAS. As mentioned before, a DBMS is viewed as a server by MDAS. A given DBMS can potentially manage several different databases each of which can contain multiple data sets. By registering the DBMS in MDAS, one is *not* necessarily registering each of the databases of that DBMS (though this could be provided as an option). However, each individual database, or each table within a database, can be registered as a data set.

One aspect of the metadata stored for data sets would deal with how to access the data set. This will differ depending on whether the data set resides in a system that provides file-based I/O to that data set, or whether the data set resides in a relational DBMS with SQL as interface, etc.

Method Registration: Each program, utility, macro that needs to be accessible by MDAS clients, can be registered as a method in MDAS. A common method description is used so that a common set of metadata can be maintained for all methods (of course, each class of metadata may have additional metadata describing it). The metadata common to all methods includes a description of the method, specification of inputs, and outputs.

Client Interface Service

The client interface service is the one that allows interactions to occur between clients and the MDAS system. These consists of requests for access to data sets, requests for execution of methods, and scheduling of specific resources within the system. As indicated in figure 4.1, this set of services may, in fact, be provided by several modules within MDAS. As indicated in that figure, each client type (Web, Application, DBMS), may have a separate server module to provide the interface service.

Client Authentication: As part of the Client Interface Service, client authentication is used to ensure that each client making an MDAS service request is *bona fide*. Metadata stored at client registration time can be used to aid in the authentication process.

Server Interface Service

The server interface service allows the actual allocation of servers to tasks and supports the execution of processes on various servers. These are the processes that get work done on behalf of client requests. This can include execution of a method on a parallel engine, retrieval of a data set from archival store, display of a data set on a visualization engine, etc.

Server Scheduling: After it receives a set of client requests, the MDAS system can schedule the computation since it now knows about the data set and methods being used and the servers needed to process this request. In addition, since MDAS maintains metadata related to servers, it also knows the current state of each server. This information is used to efficiently schedule the computation in the system.

Data Formatting and Data Transfer Service

An MDAS request may require the application of several processing steps to one or more data sets. In general, this requires the ability to move data sets from one processing step to the next and, possibly, reformatting data to suit the needs of the various intermediate steps. This may be supported by built-in methods available to MDAS or by explicit steps specified in the original request itself.

Data Formatting: This is a specific service provided by MDAS where a set of built-in methods are available for reformatting data between a variety of formats. If reformatting is needed for a particular client request, MDAS is able to refer to its library of methods and arrange for the appropriate conversion. Data reformatting may be required for a variety of reasons. At the lowest level of data representation, reformatting may be needed to convert the byte ordering of data. At a higher level, it may be required because, for example, a parallel data set has been partitioned in one way but is required to be processed in some other way, or it has been partitioned into N pieces but must be processed on M parallel processors, where N and M are different. Based on the metadata available, the MDAS system is able to determine if any reformatting is needed and is able to call upon the various methods available to the reformatting services to achieve this.

Data Transfer: In servicing a client request, it may be necessary for MDAS to schedule tasks on multiple different servers and to transfer data sets between servers. Given the location and format of the source data set and the location and format of the destination data set, the data transfer service ensures that this transfer can occur as efficiently as possible. In addition, it also provides for the transfer of data sets to the client programs, if needed. Part of this service is to support standard parallel transfer protocols such as MPI-IO or its extensions.

4.2 Describing Metadata

The metadata architecture discussed in sections 3.1 and 4.1 requires a record structure that is both rich in content yet concise for communication and transaction efficiency. The minimal level of "rich" content required is system-level metadata regarding ownership, access permissions, applicable methods, lineage, applicable names for the object, its storage location and attributes, and some definitive information about the object's *format*; i.e., the type, form, and binary format.

4.2.1 Considerations on Descriptions of Format

Several approaches to the latter have been explored in recent years. A popular approach for Web-based systems are MIME-types [6]. An object-oriented (OO) approach which requires linking to specific class libraries has been implemented in OO DBMS systems with CORBA [7]. Researchers interested in mass-storage systems have sought to define metadata standards for file storage, query, and retrieval [9]. Efforts to define detailed, all-encompassing metadata structures for object-relational DBMS systems are also underway [3]. HDF [5] and NetCDF [15] are successful file-based approaches used by researchers in scientific computing.

Our approach is a hybrid of these efforts. To limit the size of actual metadata records, we do not attempt to describe the details of an object's format, but instead introduce metadata fields to reference representative *names* of formats. These names may be user or system defined. It is anticipated that they will often be common file formats or mime types. In addition, these fields can be used by a DBMS to reference user-defined schema or tables. This approach allows a DBMS or data-mining agent to produce cross-reference tables of data sets, methods, resources (and users!) which are "format" compatible.

As discussed in section 3.1, "format" is a multi-parameter concept which includes the ideas of data types, data form (structure in memory or other storage), binary formats, etc. Some self-describing file formats are contain all "format" parameters in an internal header. Thus, we found it necessary to support multiple *name fields* for the specification of an object's format. For some objects, one field may be sufficient. A description of our schema design is given in section 4.2.2.

For data-mining—particularly on parallel storage systems, multi-parameter format specifications are essential. It is not enough for the data-mining application to discover that a large image of interest is stored on a particular system: the application also needs to learn details of the storage layout, the logical data layout, and the numerical types which compose the image. This permits the application to create one or more buffers at run-time which can perform the parallel file transfer and manipulate the data in a meaningful way. We refer to this kind of transaction as "3rd party data handling" since the data-mining application did not originate the data set and had no previous knowledge of its structure. Standards to support these kinds of 3rd party data type descriptions are under development by the MPI Forum [12].

4.2.2 An MDAS Schema

Each time a data set is stored, retrieved, or re-written, the system is capable of collecting metadata on that transaction. Some possible metadata records for data set transactions include: (a) data set id, (b) user name or id, (c) time and date, (d) transaction type, (e) client, and (f) client method (application program). Obviously, one cannot store such information for every retrieval of data sets on a public server: the metadata would rapidly outgrow the storage size of the object in many cases. Hence, summaries or spectral-based schemes are desired. For example, the system could log records for every transaction, then periodically summarize into a spectral-based records.

In contrast with file-based user environments we have expanded the notion of file ownership. In the MDAS schema, a data set can have multiple owners: an owner is simply a user that has explicitly added a data set to their personal user space. Multiple ownership is a storage efficiency: it eliminates most needs for data set duplication.

Permanence is another new attribute for data sets not usually associated with file systems. The permanence is a floating point value in the [0,1] interval which weights the probability of keeping (opposed to purging) the set in the repository. For example, data sets installed by system personnel for permanent public consumption will have a permanence of 1.0. Scratch data sets created by the system during intermediate processing will have a permanence near 0. Data sets with intermediate values will be kept (or purged) depending upon their access frequencies. The thresholds associated with permanence values should be site selectable.

Data set ownership is initialized automatically when a data set is created or first imported. At this time, access constraints (analogous to file permissions) are also designated. A user can choose from a range of constraints: private, group(s), or public.

Users can also add themselves as an explicit user (owner) of an existing data set through any of the MDAS clients. (For this to occur, the data set must have suitable access constraints.) This has long-term effects on the permanence of an object (data set). For example, if a user has made a data set readable by a specific group and one or members of that group have explicitly added the set to their user space, the data is not erased when the original "creator" of the data set deletes it from their user space. Instead, it remains in the system subject to the constraints of its present "owners".

Further, a data set with public access permissions might be owned by one user but (unknown to the "owner") be used by many. In this case, the metadata for that data set will develop significant retrieval history distributions. When the owner "removes" the data set from their personal user space, the access state of this object will be detected and placed into public ownership with a nominal permanence.

In an alternate scenario, the owner of a public readable data set might decide to lower its access scope (e.g., to private). If the other users of the data set have not explicitly added the object to their personal user space, then their ability to access the data is terminated. However, if some user has added a data set to their personal space and then the original owner eliminates public access, the new "owner" retains the set in their space. Further, since the data set was acquired with public access constraints, that constraint remains in the new owner's space.

Should demand for an object with low permanence vanish over time, the system may purge it or expel it to off-line tape storage. However, the metadata on an object need not be purged. Instead, the metadata "status" field can be set to an appropriate value to reflect the storage status of the object. Individual sites should be able to specify thresholds for purging and expulsion.

Global access and permanence fields are provided in the MDAS data set schema. This permits rapid processing by the system during retrieval requests or automated management tasks. The global values are simply reset during any specific constraint requests by owners. (Guarding against race conditions will be an important implementation consideration.) For example, if one owner reduces the access constraints on a data set from public to private, but the access constraints set by a second "owner" remain "public", the overall (global) access constraint will also remain "public".

Note that in a general sense, the data set transaction records implement a backward-linked list over the set of all data sets. It is an open question whether forward links would be useful or even possible. For example, tracking "what will be produced" in the case of retrievals could be very difficult in interactive, event driven scenarios.

Much of this information is of great use to the system scheduler. Although current DBMS products track most of these records automatically, implementation and exposure to external systems is a non-trival task.

MDAS Schema Prototype

The following list enumerates the metadata descriptors we currently deem necessary for general state specification of objects in a massive data analysis system. In such a general schema, the specified object type dictates the context of any given field. Not all fields apply to every object type or classification.

When implemented in a DBMS, this list would be the columns of a specialized table (see section 4.3.2). When transported to clients and servers, it can take on a number of in-memory representations including a class library in OO methods, a struct or compound array in C or Fortran, and a variety of other representations in other languages. Transportability to various processing domains and a useful set of operations on the metadata in that domain are central issues in the schema design.

MDAS_SCHEMA

General

- MDAS_type. One of user, data, method, or resource.
- name. The object's name.
- aliases. A list of alias names.
- version. Version of the object, for coherency.
- context[6]. Searchable fields related to application contexts.
- summary. Summary or abstract, if any.
- documentation. Link to metadata for object documentation, if any.

Storage

- storage_location. Address or path to storage location.
- storage_state. Type of storage system; e.g., file system, archive, DBMS, etc.

- content. Opaque handle to object.
- size. Object size, in bytes.
- storage_format. Secondary storage attributes, including storage class or applicable I/O methods.

Structure

- format_protocol. A keyword identifier; e.g., MIME, HDF, NetCDF.
- type. The actual type corresponding to the format_protocol.
- form. A keyword identifier or link to datatype descriptor.
- input_format. For methods, the format of input data sets.
- output_format. For methods, the format of output data sets.
- language. As applicable.
- state. Spatial state of object; e.g., source vs. compiled.

Lineage

- environment. The resource environment that generated the object.
- parent. Data set lineage.
- birthday. Lineage, origination timestamp.
- update. Timestamp of last update.
- derivation. Method lineage.
- start_count_time[8]. Timestamps maintained by statistics collection subsystem.
- retrieval_count[8]. Counts of object retrievals since the timestamps given in start_count_time. The 8 counts range from periods of seconds to decades. Updates are queued for hourly or daily processing. The coherence of this data is only guaranteed by the central metadata repository.
- retrieval_hist[8]. A lineage of object retrievals or accesses, compressed into stochastic spectra. The 8 spectra range from periods of seconds to decades. or daily processing.
- write_count[8]. Counts of object "writes" since the timestamps given in start_count_time.
- write_hist[8]. Lineage of object "writes", compressed into stochastic spectra.

Access

- global_access. Global access constraint.
- global_permanence. Probability of not being purged.
- owner_count. Reference counter to the number of "owners".
- owner. User or group that "owns" object. By default, the system owns any object with an owner_count of 0.
- read_constraint. "Read" permissions for the object.
- write_constraint. "Write" permissions for the object.
- owner_permanence. Probability of not being purged by user.

4.2.3 Catalog Design Considerations

The design of the metadata schemata for MDAS requires careful consideration since most of the interactions will require querying the metadata database. The definition of 'metadata' with respect to Massive Data Analysis System is domain-independent and are defined for the entities that are managed by the system. In the MDA System we identify five kinds of entities for which metadata need to be maintained:

1. *Data elements* can be from a primitive element such as an attribute-vale to a file, database or even MDA system.
2. *Resources* such as computational platforms, peripherals, storage systems, communication devices, networks, etc.
3. *Users* who can be single or groups and may form a hierarchy of users based on their needs, expertise and security considerations.
4. *Processes* that can range from elements in a library, to packages, to daemons, and
5. *Usage characteristics* for which metadata is needed for statistical, accounting and scheduling purposes. Usage characteristics involve all the above entities and are developed to achieve certain goals. Hence, we have included them as a separate entity managed by the MDA system.

The definition of metadata is as follows: The metadata of an entity consist of its characterizing attributes that can be used to :
(A) (possibly uniquely) locate the entity,
(B) access the entity, and
(C) compute with the entity.

The major portion of metadata will be devoted to the locator functionality, which provides different means of locating the entity in an information space. Locator-specific metadata can come from portions of the element (implicit metadata) itself and/or will come from other parametric values (explicit metadata) that can aid in locating the element. (Eg. the contents of a document can form part of implicit metadata whereas its name and the semantics of the contents form part of explicit metadata). The access metadata provides information about how to transport the entity (if it is transportable) or how to communicate with the entity if it is a process, resource or user Transportable entities may exist in replicates, fragmented over a network, or possibly both. Transportable modes may allow parallel transport, fragmented transport and possibly subset transport (i.e., only interesting portions are transported). The compute-oriented metadata in the case of data elements, provides information about how to transform and present the data to various processes, users and resources. In the case of users, resources and processes, the compute-oriented metadata provides information about the characteristics of inputs and outputs to these elements, support structures needed by such elements for effective computation (eg. visualization capabilities in case of humans).

The schema given above is not exhaustive but provides a glimpse of important characteristics and usage of the metadata table. A sample metadata table for *data elements*

Key1	**Key2**	**AppDB**	**content**	**loction**	**method**	**format**	**lineage**
myobj		SD1	LO1	ilustra			
mytab			DB1				
mydat			F1	hpss		netCDF	
ds1		SD2					
ds2		SD3					
appl		SD4					
rscl		SD5					

Table 4.1: Example catalog table and entries.

is shown in Table 4.1. Keys 1 and 2 contain information that can be used to identify and access information. Key 1 gives a name (possibly globally unique with respect to the MDAS) that can be used as an access handle. Key 2 points to other tables that contain data element- and domain-specific information about the data set. For example, in the case of textual documents, there may be tables that contain an inverted index on the contents of the documents, and possibly information gleaned from the titles, section names, captions and other significant parts of the document. These tables are used during access to find whether the given document(s) conforms to the search characteristics provided by the user.

The SDi's are referential pointers to information in other tables that contain metadata that are particular to the data element type. For example, *mydat* is a file stored and managed by an (possibly remote) operating system. The SD table for that data type can contain information such as the remote host address, port number for connecting to the file system, access method (eg. ftp, UniTree access, HPSS access), buffer sizes needed to transfer and store to obtain optimal data transfer, flags and modes necessary for the transfer, authentication information, remote file handle, data format of the data stored in the file, data format for storing in local memory, required transformation processes for converting the formats, etc. The SD tables also can contain details about the ownership of the data sets, creation lineage (i.e., which software package created it using which parameters, etc). Similar metadata for other types of data elements (such as database tables, persistent objects, large objects, etc) will be stored in other SD tables. The other attributes in the table given important details about the data elements that can be used for speedy access of the data element.

Figure 4.3 provides view of how the Catalog Data Table (CD) and the Specific Data Tables co-exist in the MDA System. Note that separate SD tables exist for users and applications.

There are six major problem areas that need to be of concern in any system that uses metadata. We provide below a brief description of these areas.

Generation Issues - Issues in this area deal with what elements of the data and its parameters forms a necessary and sufficient set of metadata that can used to locate, access and compute with the data. This area is also concerned with how one can go

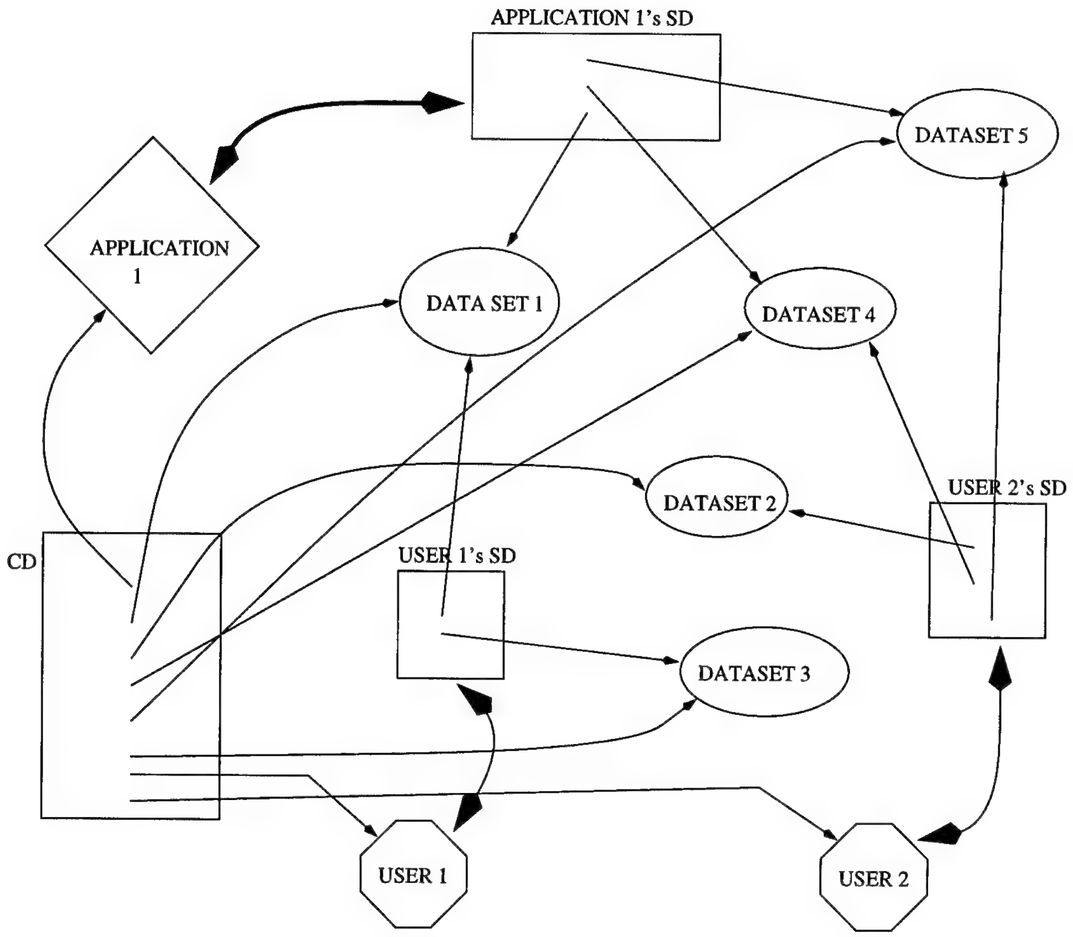

Figure 4.3: Relation between MDAS CD and SD Tables.

about capturing the necessary metadata in an automatic fashion.

Representation issues - One also needs to concern about how the metadata is represented at various stages in the MDA system. For example, one needs a representation scheme with which user or automatic programs can insert and update metadata, one also needs a representation scheme for transporting metadata across systems and that can be read and written in a uniform system-independent manner. One also needs a scheme for internal representation of metadata at the memory level that is system- and language specific. One also needs a multi-level representation. One can view a hierarchy of metadata for the same object - from a coarse-grain containing a few metadata information to a fine-grain version containing all metadata about the object. Such a system will be useful for accessing data based on available knowledge about the data.

In our design, we plan to develop a *Meta Data Markup Language (MDML)* as a common distribution format that can be used for updates, transportation and as a common format for translating into storage and memory representation of metadata structures for language- and system-specific functions and type-specific values (eg. video). In our next phase, we plan to concentrate on the development of such a language.

Storage Issues - Since the quantity of metadata can overwhelm any system, one need to consider issues about multiple complex memory hierarchy of storage, local and remote location of metadata, distribution and fragmentation of the tables, and finally replication of metadata.

Maintenance issues - One also need to be concerned with issues of fault-tolerance, schema evolution, versioning, and update problems. Since the metadata schema and the metadata itself would be evolving over time, considerations for extensibility should be central to the design.

Retrieval Issues - Since metadata can reside on different platforms, possibly fragmented and replicated one needs to have a fully-transparent retrieval system that takes care of issues such as optimization, cacheing, sharing, authentication and security, guaranteed levels of value-consistency, and active retrieval (eg., camp out for certain metadata).

Legacy metadata Issues - Finally, since there exist vast amount of domain-specific metadata one needs to give careful thought to the design of the system in order to allow assimilation of existing metadata in the new framework.

We consider that the projected outcome of A well-defined project will solve the following problems:

- identify a metadata hierarchy and forms of metadata,
- design a MDML language,
- develop data structures at the internal levels for interoperability and intraoperability of processes using metadata,
- identify and describe operations on metadata and their transformations,

- design a middle-metadata layer that can manage metadata and negotiate between application and the data layer for finding, accessing, modifying and using data,
- develop, as a feasibility study, a prototype system using the middle metadata management layer.

4.3 A Software Architecture For Metadata Messaging

To support the functionalities described in section 4.1, a system level middleware layer is required to provide efficient messaging, caching, and coherency of metadata "records" between possibly distributed service components in the data analysis system. To build a coherent distributed system, a common set of primitives are necessary for sending, receiving, reading, and writing metadata records—along with in-memory and out-of-core storage representations for the metadata. For portability, the design needs to consider the plethora of computer language and architectures likely to be used in the MDAS environment. In this section we discuss the basic middleware architecture and the representation of MDAS metadata in the MDAS metadata server.

4.3.1 A Metadata-Centric Design

The basic software architecture is given in figure 4.4. The general approach is to build interface libraries on top of a common set of primitives. From these libraries, client- and server-specific software modules can be built to interface with standard technologies such as Web services, applications, and DBMS services.

The MDAS software primitives define in-memory representations of MDAS metadata for common programming languages. These include Fortran, ANSI C, C++, and Java. Three "on-the-wire" formats are also specified for metadata communication via ODBC, MPI, and TCP/IP sockets. For out-of-core metadata record caching and storage, an SGML text format plus a binary dump format are also defined. In addition to data structures for MDAS metadata, the MDAS primitives also include some basic data structures for authentication mechanisms and "user space" data.

To ease the software engineering task of implementing and maintaining these 24 representations (4 languages $\times$(1 in-memory +3 on-the-wire +2 storage)), a single representation is maintained in a specialized SGML source document. A suite of software development tools then "compiles" the SGML template into source code for each language. An alternate design would be to develop one representation in a single programming language and then link it as a foreign library object to the remaining dialects. This however leads to portability problems. It is far more tractable to develop native code which works with native messaging protocols than to maintain a database of foreign language symbol table conventions for each compiler vendor in each language.

Built directly upon the MDAS primitives is an elementary library layer titled "MDAS access and user space libraries" (see figure 4.4). It provides basic operations on the basic

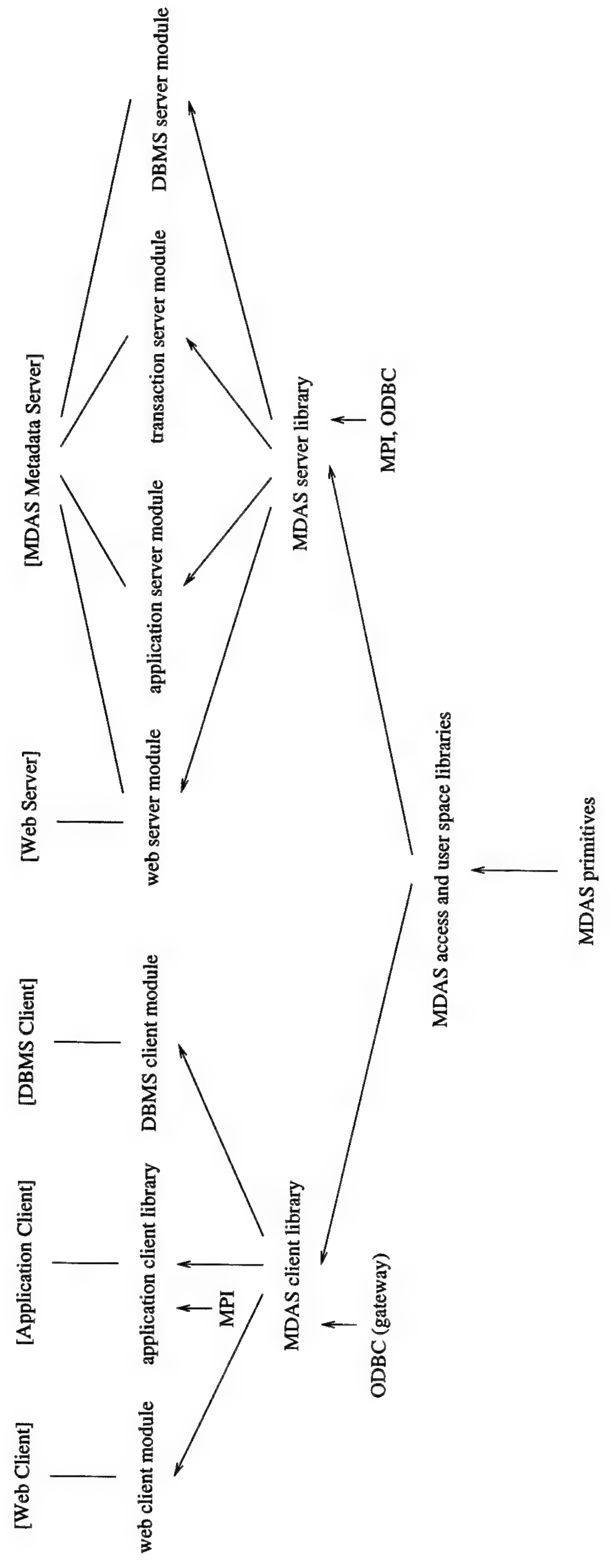

Figure 4.4: Massive data analysis system software hierachy. Services in the MDAS architecture (figure 4.1) are shown in [brackets]. 26

MDAS data structures. This library includes methods in each language for buffering, comparing, and manipulating the contents of each structure. Methods for translating between various storage and on-the-wire representations are also provided.

Two higher-level libraries are constructed from the access/user-space libraries: the "MDAS client library" and the "MDAS server library". Each is designed to meet the respective needs of clients and servers. The client library links to ODBC gateway software: either vendor-provided ODBC libraries or a custom gateway for those platforms that currently do not support ODBC. ODBC is required in MDAS for whenever direct communications with a DBMS is required.

In addition to ODBC, the MDAS server library incorporates MPI [12], an emerging standard for parallel messaging and I/O. This permits any MDAS "service" to communicate and otherwise transport data in parallel between other service providers, MDAS storage resources, and MDAS application clients. Thus, the MDAS application client library also links to MPI. At present, Web and DBMS clients have no apparent need for parallelism since these software architectures have no internal parallel mechanisms. When these clients require data residing in the parallel memory or storage of an MDAS server, the data must be agglomerated and sent in a sequential stream to the client. MPI contains a rich set of tools to perform this collation.

4.3.2 A DBMS Meta-Schema Design

Given the heterogeneity of objects and resources that are recorded in the DBMS it is impractical to capture their description in a single, universal schema. However, as has been observed before, it is possible to extract a uniform, homogeneous component from the description of all MDAS objects (metadata). Therefore, we propose that the MDAS metadata be divided into two categories: one that captures the homogeneous part of the metadata and the other that captures the heterogeneous, object class specific part.

The first category of metadata, called Common Data (CD), specifies the general state of MDAS objects and is stored in a relational table that uses the schema described in section 4.2. Thus, every MDAS object has an entry in the CD table and is generically described using relevant parts of the schema.

The second category of metadata, by nature, must be specific to a class of objects and there can be many such classes in MDAS. We call this metadata Class Specific Data (SD). The metadata schema definition for the SD of any particular class of objects is a user/application task. MDAS applications can either use an existing SD schema or create a new SD schema to store the class specific data of the objects they create. The class specific data is stored in relational tables using the application defined SD schema.

Besides storing metadata the CD table and the SD tables also store handles or pointers to the objects they describe. These object handles (possibly in conjunction with other metadata) have enough information that allows applications to retrieve the objects from their storage servers.

Although from the description above it would seem that the CD and SD parts of an object's metadata are independent, their usage and maintenance is not independent of each other. The dependencies exist because applications may try to retrieve objects by searching across both forms of metadata (requiring a join between the tables) and also because of data integrity requirements. In order to maintain these dependencies we use several standard relational DBMS features such as catalogs, primary–foreign key relationships, and triggers.

The SD schemas created by applications are tracked in the DBMS using a class table, i.e., the class table contains an entry for every SD table created in the DBMS. The class table allows applications to query the DBMS for availability of appropriate SD schema for storing the class specific data. Furthermore, it also allows for the checking of the primary–foreign key dependency requirement between the CD table and a newly created SD table through triggers. The primary–foreign key dependency requirement between the CD and SD tables ensures that queries requiring joins between them can be performed efficiently and also that the data integrity is maintained.

In summary, the CD and class table of the MDAS database are created when MDAS is first installed. A generic description of all MDAS objects is stored in the CD table. Furthermore, if applications so desire, they may store object class specific data in SD tables (which may have to be created and registered in the class table). Thus, it is possible that there are entries in the CD table that do not have any corresponding entries in any SD table. However, for every entry in a SD table is an entry in the CD table.

4.4 Application-Level Design Specifications

A massive data analysis system is by nature a remote system: it is a service to connect with. This is true whether the application is running on-site or across the continent: it still must interact with the MDAS services via network connections. Once communication is established, a variety of services might be available. Thus, an application needs some mechanism to discern or otherwise choose which service is desirable for the task at hand. In the case of I/O, applications can have special needs in the delivery of data and the recognition of storage formats. Finally, the security of the users connection and data transmission is important. All of these concerns need to be accounted for in the design of an API for MDAS.

4.4.1 System Access

A service can involve one-sided or bi-directional communication. The usage model for bi-directional communication is: open, request, response, close. A one-sided service can be stereo-typed as: open, request, close—where "request" is often a read or write operation. One-sided services can be characterized as bi-directional communication with no response.

Regardless of the communication model, users associate services with "connections". Connections are established through service protocol and often the specification of a service address or location. For example, a host name or IP address is used to specify the target service provider TCP/IP access protocols such as telnet, ftp, etc. In computer programs, protocols such as open() are used to establish connections with I/O device drivers.

Access to services are often complicated by security considerations. When a multi-user O/S provides storage device services to a user, security is maintained through file permissions. (This is not true of many PCs where security consists of locking the computer in a room.) In a multi-user O/S, most security protocols rely on the fact that the user 1st logged into the system with a password prior to starting any tasks. In general, there are a variety of service security models: single user model (as in PCs), dedicated network connections (single user network), password challenge (telnet model), "hard-wired" or pre-set software passwords (software equivalent of dedicated network), friendly host tables (Unix model), and tickets (kerberos, ssh model).

The ticket model can superset the others, although a change in operational paradigms may be required. Under this change, a passwordless system is considered to have null tickets. A password challenge system can permit the instantiation of tickets, where the user changes from a login+password paradigm to a ticket+login model. The friendly host table model is actually another instance of pre-set software passwords and therefore can be replaced by either null or automatically generated tickets.

Large-scale service centers often have multiple services available. This is also true in commercial institutions where multiple DBMS services are present. In the DBMS world, ODBC provides a standardized mechanism for discovering resources. A low level paradigm for network resources is SNMP. For Unix and other network hosts, DNS is yet another navigation tool. Service brokers such as these are a valuable aid to users, application developers, and system administrators. A massive data analysis system needs to maintain such an entity for the same reasons.

A simple access API

The most basic of access concepts is resource discovery. MDAS provides this through a multi-stage inquiry protocol which first requests a list of available services from a known location, then queries the list to determine what (if any) services are of interest. The two functions are `MDAS_INQUIRE` and `MDAS_SERVICE`.

```
Notation:
IN      means input variable
OUT     means output
IN/OUT  means modified

 MDAS_INQUIRE( services, loc )
      OUT     services - a result set, containing service
                         info structures for available
                         resources.  An info structure is a
                         set of {key, value} pairs.  A set of
                         utility functions for info structures
                         is provided.
      IN      loc      - {key, value} pair(s) indicating
```

```
                                 location of MDAS service broker
                                 (null for default)

MDAS_SERVICE( service, sel, smode, services )
     OUT      service  - specific info for an MDAS resource
     IN/OUT   sel      - index into services set
     IN       smode    - selection mode; e.g., prev or next
     IN       services - result set from MDAS_INQUIRE
```

Once a service is identified, the user will desire a connection. Of course, some users will know apriori what service is desired and store this information instead of calling `MDAS_INQUIRE`. MDAS provides `MDAS_CONNECT` and `MDAS_DISCONNECT` to establish and terminate connections. In addition, MDAS requires the use of `MDAS_TICKET`, a generic interface to security protocols. Applications are required to call this function to obtain a ticket for `MDAS_CONNECT`. If no supported security protocols are available on the users system, they may either connect as an anonymous user or download appropriate security software from the MDAS site.

```
MDAS_TICKET( ticket, user, host, protocol, service )
     OUT      ticket   - a ticket structure (TBD)
     IN       user     - the desired remote user name
                         (null for default)
     IN       host     - the desired remote host
                         (null for default)
     IN       protocol - the desired security protocol.
                         Some clients might have multiple
                         protocols.  (null for default)
     IN       service  - specific info for an MDAS resource

MDAS_CONNECT( servh, service, ticket )
     OUT      servh    - a service handle; e.g., socket
                         descriptors and handle, or an MPI I/O,
                         HPSS, or ODBC server handle, etc.
     IN       service  - specific info for an MDAS resource
     IN       ticket   - a ticket generated by MDAS_TICKET

MDAS_DISCONNECT( servh )
     IN/OUT   servh    - an MDAS service object.
                         Value is null or invalid after call.
```

4.4.2 Data Handling

The MDAS must support file-based applications, DBMS services, and Web technologies. Therefore, the data handling environment needs to incorporate functionalities from multiple I/O paradigms. Table 4.2 lists those currently under study by the MDAS project. A

Functionality	Paradigm
Local/Remote File Put/Get/Move/Copy	HTTP, FTP, O/S
Open/Read/Write/Close	File I/O
Coordinated parallel read/write	Parallel File I/O
Stream Object In/Out	OOP
Query/Select/Fetch/Extract/Update/Store	DBMS
Pipe To/From	O/S
Parallel Messaging	MPI
Control directives for I/O stream	O/S
Synchronous and Asynchronous operations	O/S
Data Caching and Hierarchical Storage	O/S

Table 4.2: Data handling functionalities

standard API exists that will support each of these functionalities[1] in current language and O/S environments. With the exception of FTP, none of these are designed to interact with remote resources and services. Implementing these in the MDAS environment then requires an interface that can initialize file units, file pointers, stream objects, and the like for existing I/O interfaces—but which reference data streams on remote MDAS services.

For example, consider the use of a large, legacy Fortran program with standard Fortran I/O in the MDAS environment. The user would prefer (perhaps insist!) that no major changes be required in the I/O portions of the code. The code could readily be accommodated by designing an `OPEN` and `CLOSE` interface which initialized Fortran unit numbers for data sets known to reside on remote MDAS resources. To use this interface, the user would only be required to insert an `MDAS_CONNECT` call to initiate a session with the remote service, and then a customized `MDAS_OPEN` call to obtain unit numbers for each "file I/O" sequence. Although this approach still limits the Fortran program to interaction with files supporting the program's data model and sequential streams which function like Fortran I/O units, it greatly expands the data storage resources the program can interact with.

4.4.3 Mixing Data Handling Paradigms

Of equal interest to the MDAS project is the fusing of historically different data access paradigms into new data handling environment. For example, a data mining application needs the functionality of *searching* for, *reading*, and *analyzing* data archives in an ad hoc fashion. To perform these actions, it needs at least 3 of the functionalities listed in table 4.2. Thus, additional infrastructure is required that permits interoperability between each supported functionality.

Two different developments are necessary to achieve mixing of data handling paradigms:

- stream conversion interface

[1]or is emerging, as in the case of parallel I/O in MPI

- an object naming interface

A *stream conversion interface* permits reading and writing with the same user buffer on two different data streams, and the "piping" (3rd party movement) of data from one stream or resource type to another. Given that MDAS maintains metadata on the data sets and the resources, the operations required to perform the data movement can be determined automatically by MDAS services.

There are some obvious limitations. It is relatively easy for a library interface to buffer data from a parallel storage form to a sequential input stream but the reverse (sequential storage, parallel read) is problematic. For example, consider a user request to read a sequential file on a remote `ftp` service into the distributed parallel memory space of a program. In this case, interfacing MPI-IO read/write protocols to a remote sequential service would not only require tremendous infrastructure development but would likely be a low performance solution. Instead, the user's program can discover (through metadata) that the data source is sequential, read with a sequential I/O method, and broadcast or redistribute the input with standard MPI library calls. For applications written specifically for the MDAS environment, MDAS supplies a set of basic read/write utilities which mask this complexity.

In order to mix query and standard I/O functionalities, an *an object naming interface* that can convert the names of data sets, resources, etc. used by each paradigm's interface is needed. For example, a data mining program would use a query-like command to obtain a list of data sets matching the query criteria. The user's program needs a means of inspecting and buffering the list, then converting any data set name of interest into a representation that can be used by other I/O interfaces.

In particular, suppose that an OO application is pre-processing images of interest using a C++ class library with a streams interface. As the application "discovers" data sets of interest, it needs to pass the equivalent of file names to the "open data set" interface. MDAS supports this by extending the standard Unix `fopen(filename, ...)` interface to `MDAS_OPEN(dataset, ...)`, where `dataset` is an object (or unique index reference) that is interoperable with the data set naming conventions in the MDAS query functions.

```
MDAS_OPEN( dh, dataset, servh, attr, hints, comm )
     OUT     dh        - a data handle, containing
                         information about the stream, etc.
     IN      dataset   - data set ''name'' returned by
                         MDAS_NAME
     IN      servh     - a service handle returned by
                         MDAS_CONNECT
     IN      attr      - read/write/append attributes
                         (null for default)
     IN      hints     - hints to the I/O server regarding
                         data access patterns (null for
                         default)
     IN      comm      - an MPI communicator object or null
                         if parallelism not required
```

4.5 Implementing an MDAS Archival Storage Resource

As part of the MDAS project, we are building an integrated database/archival storage system as an example of an MDAS server resource. Such a system would be registered as an MDAS server, and data sets stored in this system could be registered as MDAS data sets. Thus, the metadata repository would contain metadata for the archival storage server and, possibly, metadata for data sets stored in this system. We believe that such archival systems will become important resources in supporting data-intensive applications that manipulate very large data sets. In this implementation, we are building upon object-relational database technology and archival storage technology.

Currently, we are experimenting with different approaches to integrating the database system with the archival system. These approaches are described below.

4.5.1 Case 1 – No DBMS/Archival Storage integration

The database system (DBMS) and the archival system operate independently and they are both registered as server resources in MDAS. The database system stores metadata about data sets in the archival store. If an application wishes to access metadata for a data set and then read the data set itself, then it must do so by accessing the database system first and then accessing the archival storage system. Typically, this would have to be done by the MDAS client itself.

4.5.2 Case 2 – External File implementation in DBMS

In this scenario, the DBMS is able to directly perform I/O operations on the archival system, on behalf of the MDAS client. The DBMS offers a set of API's which provide a file I/O interface to the data sets stored on the archival system. Thus, clients can access the data sets via the DBMS using a regular file I/O interface (open, read/write, close). In addition, clients can also directly access the data sets from the archival system, using its API's. However, the data sets are still "external" to the DBMS, in the sense that they are not managed by the DBMS. The DBMS only maintains metadata associated with these data sets. From the client's viewpoint, this implementation provides a more comprehensive view than Case 1, since the client needs to make a single connection to the DBMS to query the metadata as well as to access the data from the archival storage.

4.5.3 Case 3 – Archival Object implementation in DBMS

In this case, only the DBMS is visible to the MDAS system, as a server resource. Thus, all access to the data sets is via the DBMS. Similar to Cases 1 and 2, the database stores metadata related to the data sets, and similar to Case 2, the DBMS provides a file-based I/O interface to access the data sets. However, the data sets are now stored in the database in the form of "large objects", which are entirely owned and managed by the DBMS. A data set that is imported into the DBMS is assigned a large object handle by the DBMS. It is left to the DBMS as to how it handles the transfer/migration of data between disk and archival storage. In our prototype implementation, we provide a parameter in the file I/O interface offered by the database system, which specifies whether a particular data set should be stored on disk or in archival store. Compared to Case 2, this implementation is more tightly integrated, and provides an even more comprehensive view to the client. All data (both the metadata and large objects) are now owned and managed by the DBMS and the archival storage is part of the DBMS.

4.5.4 Parallel data streams

When data is returned from the archival system to the user application (MDAS client), it can be returned either as a sequential I/O stream or via parallel I/O streams. This gives rise to two possibilities in each of the above three cases. Parallel I/O transfers may be used to provide simultaneous access to multiple independent data objects (or files), each with its own data stream, or to access a single data object with multiple data streams. In addition, for Case 2 and Case 3 above, there is the additional possibility that the DBMS reads N parallel streams from the archive and provides M parallel I/O streams to the client, where N and M are not the same. Thus, the database acts as a buffering device between unequal numbers of streams.

4.5.5 The DBMS/Archival Storage Prototype

To gain early insight into issues associated with database/mass-storage interoperability, we are building prototypes at the San Diego Supercomputer Center(SDSC), using Postgres95[17] as the database engine and the NSL UniTree and HPSS as the mass storage systems. Postgres95 is a public domain, object-relational DBMS developed at the University of California, Berkeley.

NSL UniTree is a hierarchical archival storage system currently running in production mode at SDSC. The system is capable of storing almost unlimited amount of data. This system will be replaced by the HPSS system by the beginning of 1997.

Thus far, we have built a prototype of the external file (Case 2) and large object (Case 3) implementations with the single data stream scenario.

The External File Prototype

For the external file implementation, we created a set of UNIX-like API calls on the client side and a corresponding set of API's on the Postgres server side. Thus, a Postgres client can use the client API's to access files stored in UniTree. The DBMS server calls are used internally by Postgres95 to access files stored in UniTree. These calls may also be used by UDFs (User defined functions, which are dynamically linked to the DBMS) to access the archived files.

Following is the specification of the client API:

```
int e_create(PGconn* conn, int access_m, char *host_addr, char
*filename, int mode)

int e_open(PGconn* conn, int access_m, char *host_addr, char
*filename, int flags, int mode)

int e_unlink(PGconn* conn, int access_m, char *host_addr, char
*filename)

int e_close(PGconn* conn, int fd)

int e_read(PGconn *conn, int fd, char *buf, int len)

int e_write(PGconn *conn, int fd, char *buf, int len)

int e_seek(PGconn *conn, int fd, int offset, int whence)

int e_sync(PGconn* conn, int fd)

int e_stat(PGconn* conn, int access_m, char *host_addr, char
*filename, struct stat *statbuf)
```

```
PGconn     -- database connection descriptor, obtained by calling the
              CONNECT API in Postgres

access_m   -- Access method.  Specifies where the data set is to be
              stored.  0-UNIX, 1-UniTree, 2-HPSS
Host_addr -- Host address.  Specifies the host address where the
              archival store resides
```

On the server side, the following corresponding API's are implemented:

```
int e_create(int access_m, text *host_addr, text *filename, int mode)

int e_open(int access_m, char *host_addr, char *filename, int flags,
int mode)

int e_unlink(int access_m, char *host_addr, char *filename)

int e_close(int fd_inx)

int e_read(int fd_inx, char *buf, int len)
int e_write(int fd_inx, char *buf, int len)

int e_seek(int fd_inx, long offset, int whence)

int e_sync(int fd_inx)

int e_stat(int access_m, char *host_addr, char *filename,
struct stat *statbuf)
```

The above calls are essentially UNIX-like calls. For the client side API's, three additional parameters are introduced, viz. *PGconn, access_m and host_addr.* On the server side, the last two out of those three are introduced. These two new parameters, *access_m and host_addr*, provide the framework for a very flexible way for data storage and access. The external files can now reside on disk or in archival storage, including cases where the archival storage is on a different host machine than the database system. Even multiple archival storage systems can be supported.

All three access methods, UNIX, UniTree, and HPSS, have been implemented in the current prototype.

The Large Object Prototype

Postgres95 currently already supports large objects within the database. Thus, rather than creating a completely new set of client API's and server calls, the Postgres95 large object implementation was extended to support multiple access methods. The client API is as follows:

```
Oid lo_creat(PGconn* conn, int access_m)

int lo_open(PGconn* conn, Oid object_id, int mode)

int lo_unlink(PGconn* conn, Oid object_id)

int lo_close(PGconn* conn, int fd)

int lo_read(PGconn *conn, int fd, char *buf, int len)

int lo_write(PGconn *conn, int fd, char *buf, int len)

int lo_seek(PGconn *conn, int fd, int offset, int whence)

Oid lo_import (PGconn *conn, char *filename, int access_m)

int lo_export (PGconn *conn, Oid object_id,  char *filename)
```

The DBMS calls on the server side are as follows:

```
Oid lo_creat(int access_m)

int lo_open(Oid object_id, int mode)

int lo_unlink(Oid object_id)

int lo_close(int fd)

int lo_read(int fd, char *buf, int len)

int lo_write(int fd, char *buf, int len)

int lo_seek(int fd, int offset, int whence)

Oid lo_import (char *filename, int access_m)

int lo_export (Oid object_id, char *filename)
```

Except for *lo_import* and *lo_export*, these calls are UNIX-like calls. *Lo_import* is used to import a UNIX file as a large object and *lo_export* is used to create a UNIX file from a large object. As before, the *access_m* parameter in the *lo_creat* and *lo_import* functions allows the client to choose the access method for the large object to be created.

Using these server calls, two user-defined functions, *LocalToUtree()* and *UtreeToLocal()* have been created to provide an easier way for objects to migrate between UniTree and local file systems.

The current prototype supports the UNIX, UniTree, and HPPS access methods.

4.6 The SARA Image Delivery System Prototype

A prototype of the SARA image delivery system has been built to demonstrate the delivery of remote-sensing data across a gigabit wide-area network using a Web server front end accessing large data files stored in the Archival Storage systems (HPSS/UniTree) through a DBMS (Postgres95). The URL of this prototype is http://sara.sdsc.edu. Since the HPSS system at SDSC is not yet in production mode, the prototype may not be functioning at all time.

SARA (Synthetic Aperture Radar Atlas) is a image delivery system for SAR data from JPL. An imaging radar measures the strength and round-trip time of the microwave signals that are emitted by a radar antenna and reflected off a distant surface or object. Radar images are composed of many pixels, Each pixel in the radar image represents the radar backscatter for that area on the ground.

The SAR images are provided in discrete tracks each mapped to a rectangular region of the earth surface. Each track may contain images of up to eight channels (microwave frequencies and polarity) and the data for each channel of a track is in the form of a file approximately 50 Mb in size. In addition, a 20 Kb thump-nail (screen size course resolution image) JPEG image is provided for each track.

A SARA Web client allows a user to zero in on a section of a track for display guided by clickable maps with three level of details. When the user eventually selects a SAR track, the thump-nail JPEG image associated with the track is displayed. Since the file size and image size of each track is huge, it is not practical to display the whole track all at once. The scheme allows a user to select a small region of the track for display by clicking on the thump-nail image. In addition, the scheme allows each channel to be mapped to a color scheme and up to three channels may be combined to produce a JPEG image for display. This mapping may be optimized to highlight features of the ground such as ecology, geological formation, etc.

Currently, we have 158 SARA tracks online which includes 731 files with a total size of 15 Gb covering only a tiny portion of the earth. Replicated copies of these files are stored in HPSS and UniTree at SDSC and CalTech. In the future, we hope to provide terabytes of online data.

4.6.1 Architecture

When a Web client selects a point on the earth surface for display, the Web server queries the DBMS for data associated with this earth coordinate. The SARA metadata consists of two relations with schema given in Table 4.3. Each track is represented by a tuple in the "sara_track" relation. The "t_rectangle" attribute represents the rectangular coordinates of the track. The DBMS searches for a tuple (track) with a "t_rectangle" in which the selected coordinate lies. The "sara_file" relation represents the SAR data files. Each tuple in the "sara_file" relation represents the metadata for a channel of a single track. The "f_name" attribute specifies the name of the file. The schema allows duplicate copies of a SAR file stored at multiple locations. The "f_no_copies" attribute specifies the number of duplicates and the "f_dataservers" and "f_access_methods" attributes give the data server addresses and access methods for these files. The Track Number and File Names attributes are common to both relations and can be used for join queries.

Knowing the data server addresses and access methods, the WEB server may then use the External File API given in 4.5.5 to retrieve the data. Typically, each request involves accessing a portion of three files (combining three channels) reading 2 Mb each. Each files access involves a e_open(), a e_seek(), a e_read() and a e_close(). If duplicate copies exist, the WEB server at SDSC will try to access the file from the HPSS server at SDSC first. If the HPSS server is unavailable, it will try the HPSS server at CalTech next, and then the UniTree server at SDSC last. This flexibility of this scheme allows it easily be extended to serve terabytes of data.

```
CREATE TABLE sara_track (
   t_track    text,          -- track number, e.g., 42922
   t_name     text,          -- name of the track, e.g., Almaz 9, Russia
   t_date     date,          -- date of creation, e.g. Apr 16 01:02:03
-- 1994 PST
   t_width    int,           -- track width, e.g.2104
   t_height   int,           -- track height, e.g.8000
                             -- (2104*8000 => #pixels)
   t_channels int,           -- number of channels, e.g.8
   t_files     text[],  -- associated file names [1, .., t_channels]
   t_spectrums      text[], -- associated spectrum names [1, .., t_channels]
   t_rectangle      polygon,-- points 1,2,3,4 of track polygon
   t_thumb_file     text,   -- JPEG file name
   t_htmls          text[], -- associated HTMLs
   t_descriptions   text[], -- associated text description files
   t_dataservers    text[]  -- data servers location
);

CREATE TABLE sara_file (
   f_track          text,   -- track number - e.g.42922
   f_name           text,   -- file name
   f_size           int,    -- file size
   f_permission     int,    -- file permission as int number
   f_date           date,   -- file date
   f_time           time,   -- file time
   f_count          int,    -- file accessed counter
   f_file_type      text,   -- file type (data, jpeg, etc)
   f_media          text,   -- file type media (disk, tape, etc)
   f_spectrum       text,   -- spectrum channel (e.g. LHV)
   f_no_copies      int,    -- no of copies of the file
   f_dataservers    text[], -- data servers location
   f_local_paths    text[], -- location of file, local path for file
   f_access_methods text[],-- e.g. unitree, hpss, local etc.
   f_owners         text[], -- file owners
   f_lo_types       int[]   -- 0 == internal, 1 == external
                            -- large object type
);
```

Table 4.3: The SARA metadata schema

Chapter 5

Important Findings and Conclusions

The primary challenges in the MDAS project are (a) the integration of data management systems with archival storage and (b) end-user solutions for the replacement of the (Unix) file paradigm with a higher-order interface to data, methods, and resources. MDAS will develop robust prototype solutions to these challenges, but several general problems will remain that merit follow-on efforts. These include:

Intelligent Hierarchical Storage Systems : Current HSS technology is designed for either (a) atomic file input/output by many users or (b) general read, write, and seek operations by a few users. An internal intelligent queueing mechanism is desirable to scale general I/O capabilities to a large number of simultaneous user requests.

Software Development Environment Standards : The lack of standards in system software tools makes developing multi-platform software a tedious process. The further development and acceptance of POSIX standards for Unix will provide some relief in this area. The situation is particularly acute in high performance computing. In 1995, the NCO for HPCC sponsored a report by the System Software Tools Working Group (SSTWG) on desired standards in system software tools. Reference to this report in procurements is likely to have a major effect on vendor compliance.

Heterogeneous MPI : The MPI Forum is defining protocols which will enable the communication of data-type structures and file structures between third-party applications. Further, MPI is defining an interoperable storage description that will allow the same binary file to be read by different binary format computing platforms. These capabilities are restricted to applications running the same version (implementation) of MPI. Further, these communication and storage modes are optional to the user so that applications desiring higher-performance communication and storage protocols may have them on vendor-specific architecture. However, it is likely that no single implementation of the MPI standard will run on all platforms of interest to a particular site; i.e., there is still a need for interoperable *implementations* of MPI. This could be acheived by defining an optional interoperable communication mode in the MPI standard itself.

Advanced OO technology : Object-oriented (OO) software technologies greatly simplify the task of software engineering and hold great promise for software reuse. However, present-day OO compilers do not produce high-performance executables. Improvements to both OO languages and compilers would be of great benefit.

Dynamic Loading : A "just-in-time" compiled applet is essentially a dynamically loaded software module. Dynamic loading has existed in some Unix compiler technology for several years, but is not standard practice. It is particularly useful in data mining and analysis applications for compiled source code derived from symbolic mathematics and query languages. Dynamic unloading is equally desirable when a module is no longer needed.

Universal Resource Names : Scientific applications should be able to access data and cache it locally no matter where the data is originally located. This is equivalent to requiring a catalog or expert system with universal resource name (URN) capabilities.

Resource Discovery : Current O/S technologies do not provide adequate interfaces for resource discovery. For example, to "discover" that a particular DBMS is running on a remote platform, the user must perform manual work to find the port number and appropriate library interface. SNMP provides a partial solution. A general O/S independent mechanism for automated discovery is needed. Meeting site dependent security needs will be an important aspect.

Parallel I/O : Support for parallel I/O streams must be done within the context of emerging standards. This requires tracking the MPI2 IO effort which is examining issues related to message passing within a compute platform and I/O to external peripherals. Interoperability between MPI and non-MPI processes will require specialized software interfaces.

Distributed Computation Support : Data sets may be distributed to multiple platforms, for analysis by methods that are retrieved from a DBMS. Support for distribution of computation objects is needed.

Third-party Authentication : Methods and data sets need to validate their interoperation through an authentication mechanism that is independent of the local operating system.

Common Communication Layer : Many of the above problems could be solved by a standardized communication layer that addresses concerns across all the sectors of the computing community. At present, there are many protocols and software implementations available with limited capabilities from which general prototypes can be developed. The NEXUS system from Argonne National Laboratory is example.

Chapter 6

Implications For Further Research

The issues researched in MDAS are essential in enabling the "Distributed Object Computation Testbed" project which will build a complex document handling system on top of federated databases that access replicated archives. The integration of database, archival storage and application (Web in particular) technology promises to facilitate the manipulation of large data sets and large collections of data sets. One goal is to enable data analysis on terabyte-sized data sets retrieved from petabyte archives, at an access rate of 10 GB/sec. Current supercomputer technology supports a 1 GB/s access rate to 1 terabyte of disk. For a teraflops supercomputer with 10 TB of disk, data rates on the order of 10 GB/s will be feasible. This will require, however, support for parallel I/O streams, and support for striping data sets across multiple peripherals. Fortunately, the software technology to support third party transport of data sets across parallel I/O streams is being developed in the HPSS archival storage system. Data redistribution mechanisms for the parallel data streams are being standardized as part of the MPI-IO effort. The expectation is that the initial usage prototypes described above can be extended to support supercomputer applications that analyze arbitrarily large data sets.

Bibliography

[1] F. Berman and R. Wolski. Scheduling from the Perspective of the Application. In *Proc. HPDC '96*. ftp://cs.ucsd.edu/pub/berman/hpdc96.ps.

[2] F. Davis, W. Farrell, J. Gray, R. Mechoso, R. Moore, S. Sides, and M. Stonebraker. EOSDIS Alternative Architecture. Technical report, San Diego Supercomputer Center. 1/23/95,
http://www.research.microsoft.com/research/barc/gray/eos_dis/.

[3] Federal Geographic Data Committee. FGDC Standards Reference Model. http://fgdc.er.usgs.gov/fgdc.html.

[4] S. Fineberg. MPI-IO: A Parallel File I/O Interface for MPI. http://lovelace.nas.nasa.gov/MPI-IO/.

[5] Mike Folk and Quincey Koziol. HDF—The Next Generation. http://shemp.ncsa.uiuc.edu/NCSA/Pubs/access/96.1/hdf-tng.html.

[6] Network Protocols Working Group. MIME protocols, parts 1 and 2. http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html.

[7] Object Management Group. CORBA 2.0. http://www.omg.org/corbask.htm.

[8] S. Tuecke I. Foster. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing.* ftp://ftp.mcs.anl.gov/pub/nexus/reports/hpdc_java.ps.gz.

[9] IEEE Computer Society Technical Committee on Mass Storage Systems. Workshop on Metadata for Scientific and Technical Data Management. http://www.llnl.gov/liv_comp/metadata/minutes/minutes-1994-05.html.

[10] John F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. Technical report, Univ. of Virginia CS-96-03. 1/96,
ftp://ftp.cs.virginia.edu/pub/techreports/CS-96-03.ps.Z.

[11] Sandia National Laboratory. Sandia Joins R&D Effort To Bring Teraflop Supercomputing On-Line. http://www.cs.sandia.gov/teraflop.html.

[12] MPI Forum. MPI 2. http://www.mcs.anl.gov/mpi/mpi2/mpi2.html.

[13] MAC OS. ©Apple Computer Corporation. http://www.apple.com.

[14] Andreas Paepcke, Steve B. Cousins, Hector Garcia-Molina, Scott W. Hassan, Steven P. Ketchpel, Martin Rvscheisen, and Terry Winograd. Using Distributed Objects for Digital Library Interoperatbility. *IEEE Computer*, May 1996. http://www.computer.org/pubs/computer/dli/r50061/r50061.htm.

[15] R. K. Rew, G. P. Davis, S. Emmerson, and H. Davies. *NetCDF User's Guide, An Interface for Data Access.* Unidata Program Center, 2.4 edition, 1996. http://www.unidata.ucar.edu/packages/netcdf/guide_toc.html.

[16] Scalable I/O Initiative. Working Papers. http://www.ccsf.caltech.edu/SIO/SIOpubslist.html.

[17] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.